SOLVING PROBLEMS BY SEARCHING: UNINFORMED SEARCH

Russel, Norvig, "Artificial Intelligence, a Modern Approach" – chapter 3

Lecture outline

Search problems

Uninformed search algorithms

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Introduction

Problem solving agents decide how to fulfil a goal by finding sequences of actions that lead to desirable states.

We will consider a goal to be a set of world states - exactly those states in which the goal is satisfied.

The agents task is to find which sequence of actions will get it to a goal state.

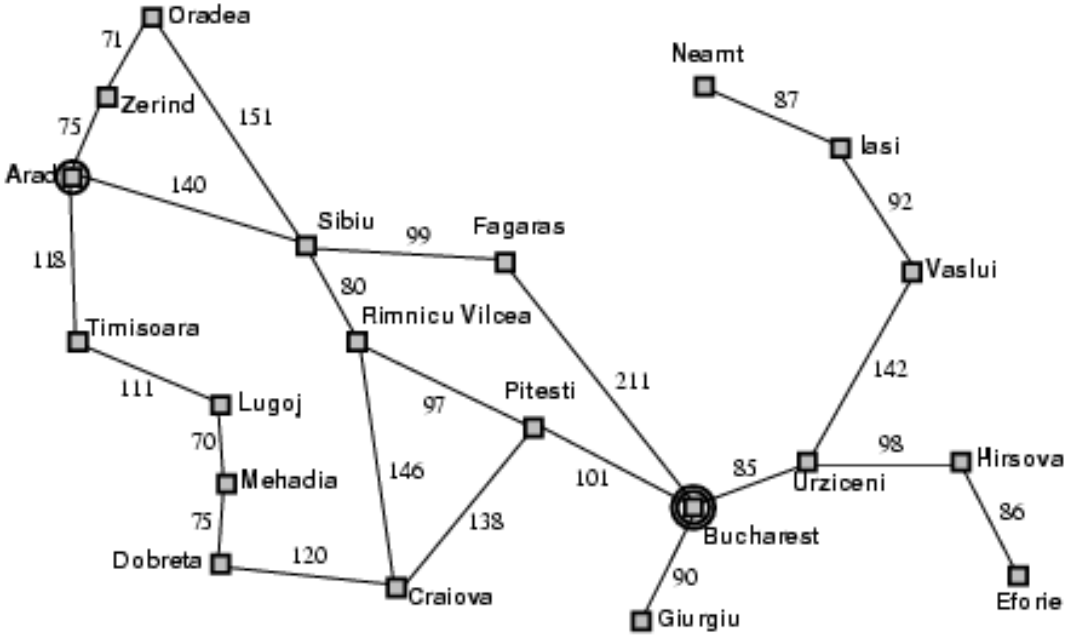
Before it can do this, it needs to decide what sorts of actions and states to consider.

Problem formulation is the process of deciding what actions and states to consider, given a goal.

Example: Romania

On holiday in Romania - currently in Arad.

Flight leaves tomorrow from Bucharest



Example: Romania

It makes sense for the agent to adopt the goal of getting to Bucharest.

Formulate goal:

- Go to Bucharest

For now, let us assume that the agent will consider actions at the level of driving from one major city to another. The states the agent will consider therefore correspond to being in a particular town.

Formulate problem:

- states: various cities
- actions: drive between cities

Example: Romania

Our agent has now:

- Formulated the goal to be in Bucharest tomorrow
- Formulated the problem as considering the actions available as driving between various cities and the states it will consider will therefore correspond to being in a particular town.

There are three roads out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.

None of these achieve the goal, so if the agent has no additional knowledge then the best it can do is choose one of the actions at random.

Example: Romania

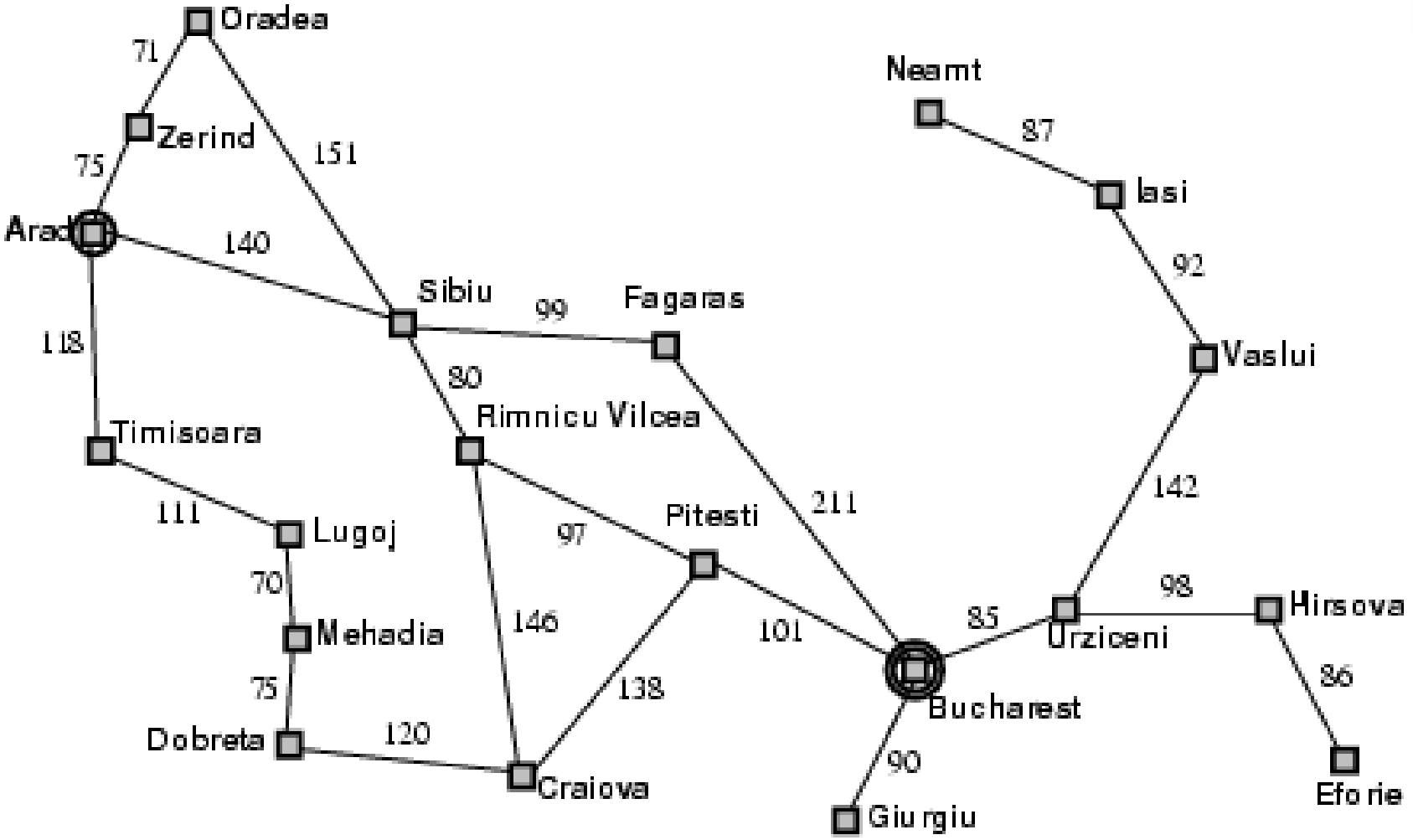
Lets imagine the agent has a map of Romania.

The point of the map is to provide the agent with information about the states it might get itself into, and the actions it can take.

The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest.

Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey.

Example: Romania



Search

In general, an agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence.

The process of looking for such a sequence is called **search**.

A **search algorithm** takes a problem as input and returns a solution in the form of an action sequence.

Once a solution is found, the actions it recommends can be carried out. This is called the **execution phrase**.

Well-defined problems and solutions

A problem can be formally defined by four items:

- Initial state
- Actions
- Goal test
- Path cost

1. The initial state that the agent starts in e.g., "at Arad"

2. A description of the possible actions available to the agent. This is often formulated as successor functions.

   – Given a state x, *S(x)* returns a set of <action, successor state> ordered pairs.
   e.g., *S(Arad)* = {<*Arad* → *Zerind*>, <*Arad* → *Sibiu*>, … }

Well-defined problems and solutions

3. The goal test, determines whether a given state is a goal state.
   - explicit, enumerated set of states e.g., x = "at Bucharest"
   - implicit, specified by an abstract property of the state e.g., *Checkmate(x)*
4. A path cost function assigns a numeric cost to each path. The agent should choose a cost function that reflects its own performance measure.
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* denotes the step cost of taking action a to go from state x to state y, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state.

Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

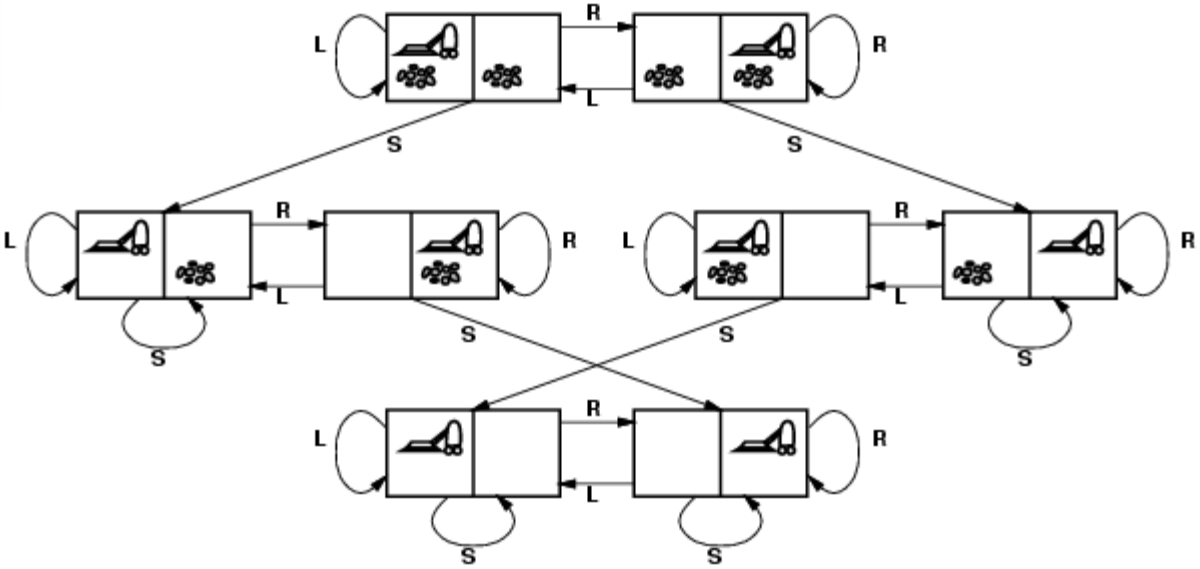
State Space and Path

The initial state and successor function implicitly define the state space of the problem - the set of all states reachable from the initial state.

The state space forms a graph in which the nodes are states and the arcs between nodes are actions; e.g. the map of Romania can be interpreted as a state space graph if we view each road as standing for two driving actions, one in each direction.

A path in the state space is a sequence of states connected by a sequence of actions.

Example: Vacuum world



States: two locations, each of which may or may not contain dirt ⇒ $2 * 2^2 = 8$ possible states.

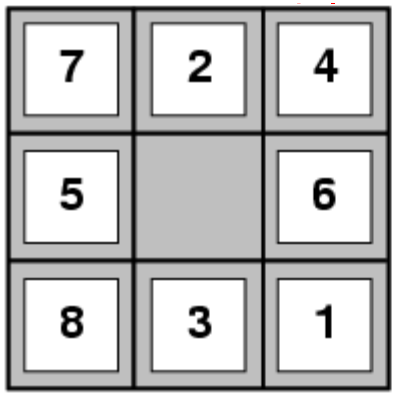
Initial state: either state can be designated as the initial state

Successor function: this generates the legal states that result from trying the three actions (Left, Right, Suck) see above state space.

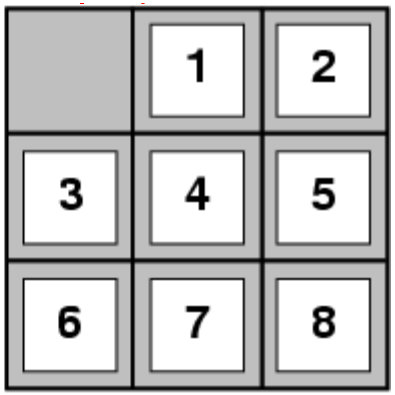
Goal test: all the squares are clean

Path cost: each step cost 1 ⇒ path cost is the number of steps in the path.

Example: 8-puzzle



Start State



Goal State

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

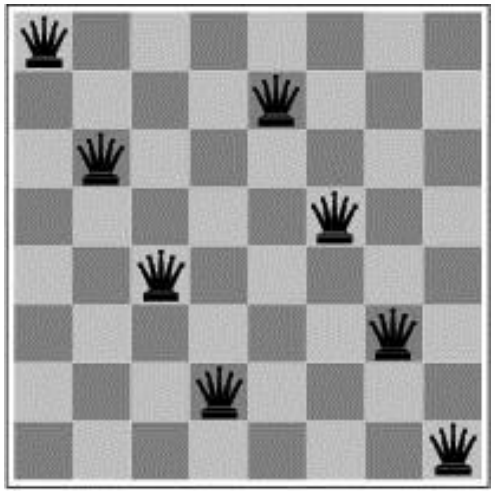
Initial state: any state can be designated as the initial state

Successor function: this generates the legal states that result from trying the four actions (blank moves Left, Right, Up, Down)

Goal test: check whether the state matches the goal configuration

Path cost: each step cost 1 $\Rightarrow$ path cost is the number of steps in the path.

Example: 8-queens



The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.

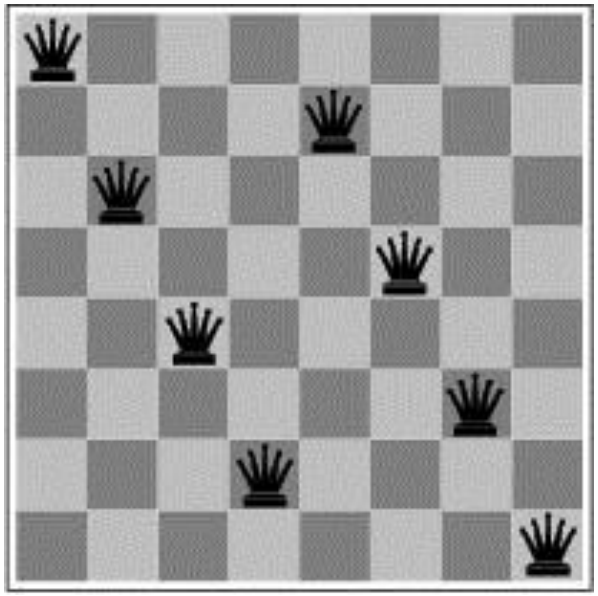
(A queen attacks any piece in the same row, column or diagonal.)

There are two main kinds of formulation:

- Incremental formulation: starting with the state description, each action adds a queens to the state.
- Complete-state formulation: starts with all 8 queens on the board and moves them around

In either case the path cost is of no interest because only the final state counts.

Example: 8-queens



Problem Formulation (Incremental)

States: Any arrangement of 0 to 8 queens on the board is a state

Initial state: No queens on the board

Success function: Add a queen to an empty square

Goal test: 8 queens on the board none attacked

Complexity: 64 * 63 * … * 57 ≈ $1.8 * 10^{14}$ possible sequences to investigate

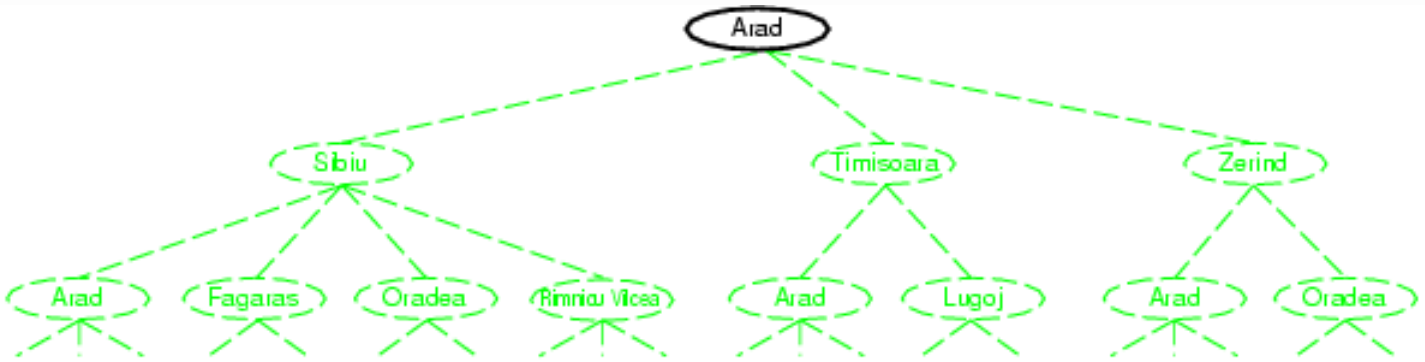
Searching for Solutions

Having formulated some problems, we need to solve them. This is done by a search through the state space.

We will first examine search techniques that use an explicit search tree that is generated by the initial state and the successor function that together defines the state space.

In general we have a search graph rather than a search tree when the same state can be reached from multiple paths.

Tree search example

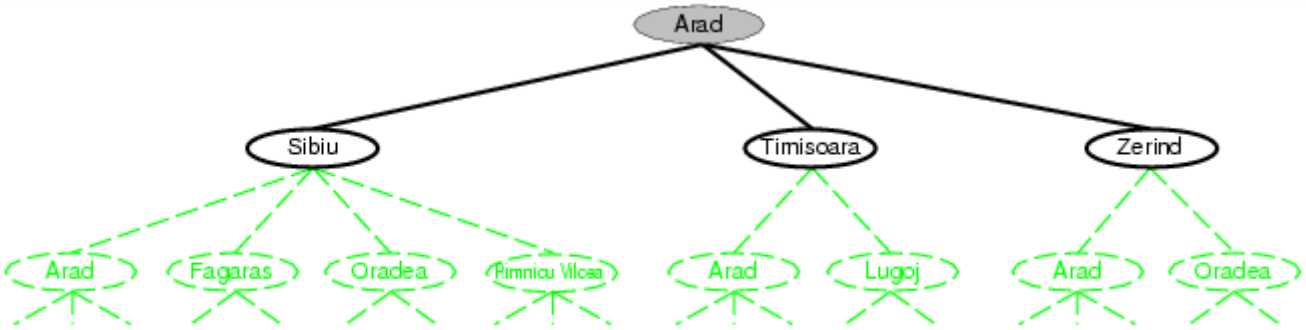


The root of the search tree is a search node corresponding to the initial state, *In(Arad)*.

The first step is to test whether this is a goal state. Clearly it is not, but it is important to check so that we can solve trivial problems like "starting in Arad, get to Arad.

Because this is not a goal state, we need to consider some other states. This is done by expanding the current state; that is applying the successor function to the current state, thereby generating new states.

Tree search example

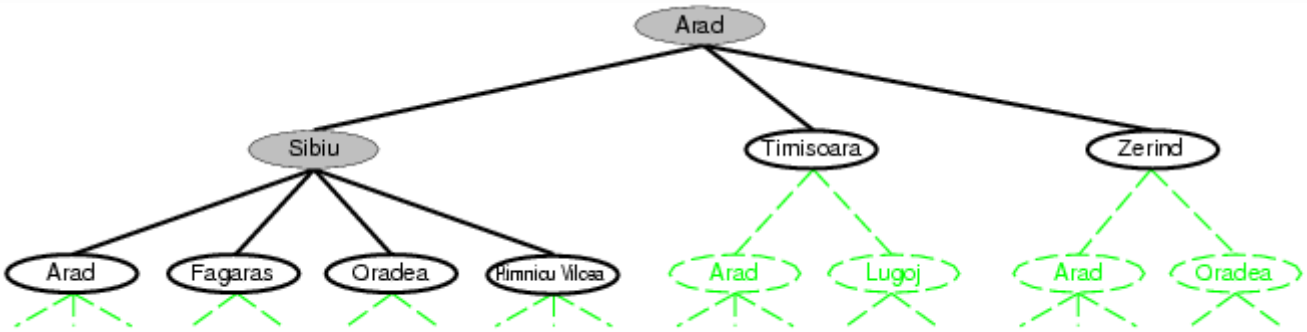


The expansion results in three new states *In(Sibiu)*, *In(Timisoara)*, *In(Zerind)*

Now we must choose which of these possibilities to consider further.

This is the essence of search - following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.

Tree search example



Suppose we choose *In(Sibiu)*. We check to see if it is the goal state. It is not.

Next we expand it to get *In(Arad)*, *In(Fagaras)*, *In(Oradera)*, and *In(RiminicuVilcea)*. We can now choose any of these four or go back and choose Timisoara or Zerind.

We continue choosing, testing and expanding until either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by the search strategy.

Tree search algorithms

Basic idea:

- simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

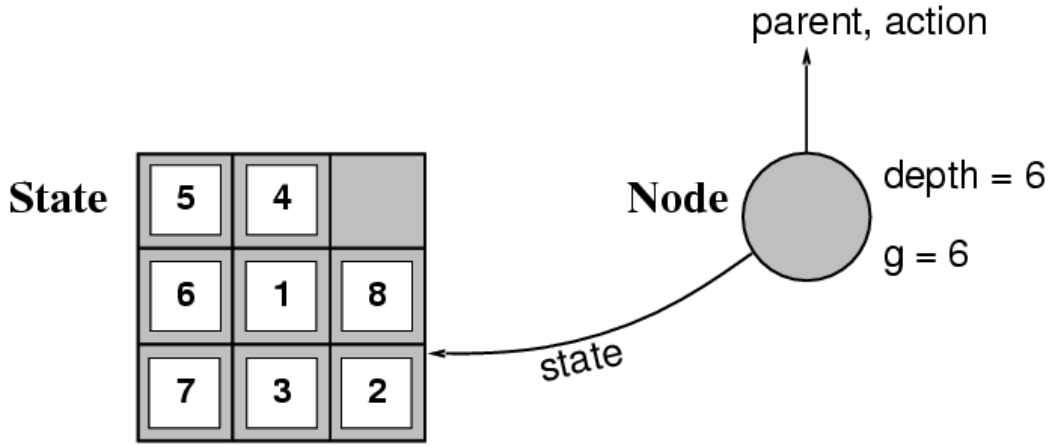
State Spaces vs. Search Trees

For the route finding problem, there are only 20 states in the states space, one for each city.

But there are an infinite number of paths in this state space, so the search tree has an infinite number of nodes; e.g., Arad-Sibiu, Arad-Sibiu-Arad, Arad-Sibiu-Arad-Sibiu

States vs. Nodes

A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(n)$, depth



The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

The Fringe and Leaf Nodes

The collection of nodes that have been generated but not yet expanded is called the fringe.

Each element of the fringe is a leaf node, that is, a node with no successors in the tree.

The fringe is normally organised as a queue as this stops the strategy function from having to look at every element of the fringe set to choose the best one.

Search strategies

A search strategy is defined by picking the order of node expansion

Strategies are evaluated along the following dimensions:

- completeness: does it always find a solution if one exists
- time complexity: time taken generating nodes
- space complexity: number of nodes stored in memory
- optimality: does it always find a lowest-cost solution

Search strategies: Time and Space Complexity

Time and space complexity are always considered with respect to some measure of the problem difficulty: the typical measure is the size of the state space graph.

Often the size of the state space graph cannot be explicitly characterized, and time and space complexity are measured in terms of

- *b*: maximum branching factor of the search tree (the maximum number of successors of any node).
- *d*: depth of the least-cost solution
- *m*: maximum depth of the state space (may be ∞)

Search strategies: Time and Space Complexity

Time complexity is often measured in terms of the number of nodes generated during search.

Space complexity is often measured in terms of the maximum number of nodes stored in memory.

Uninformed search strategies

Uninformed search strategies use only the information available in the problem definition:

- all they can do is generate successors and distinguish a goal state from a non-goal state.

There are several uninformed search strategies:

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

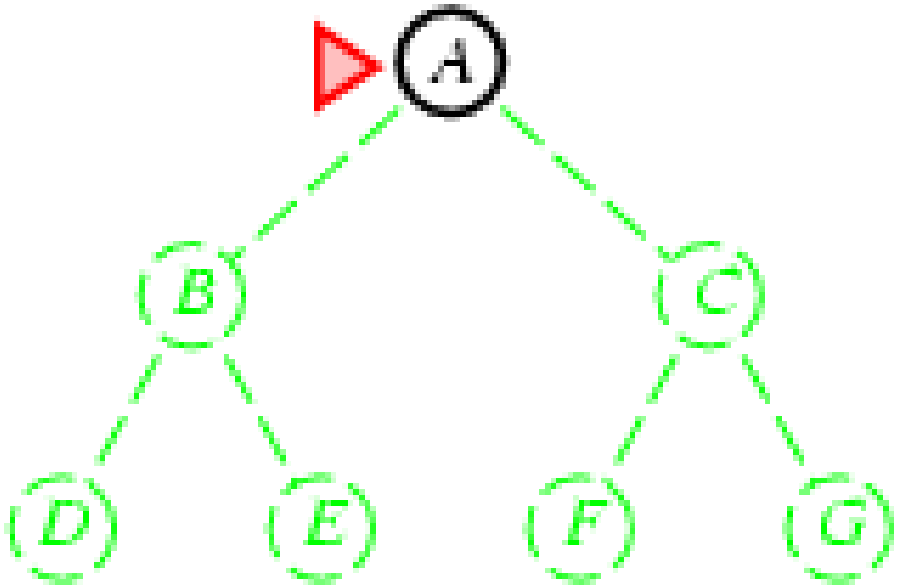
All search strategies are distinguished by the order in which nodes are expanded.

Breadth-first search

Expand shallowest unexpanded node

Implementation:

- *fringe* is a FIFO queue, i.e., new successors go at end

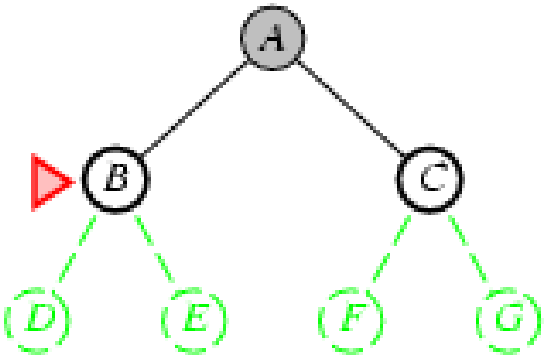


Breadth-first search

Expand shallowest unexpanded node

Implementation:

- *fringe* is a FIFO queue, i.e., new successors go at end

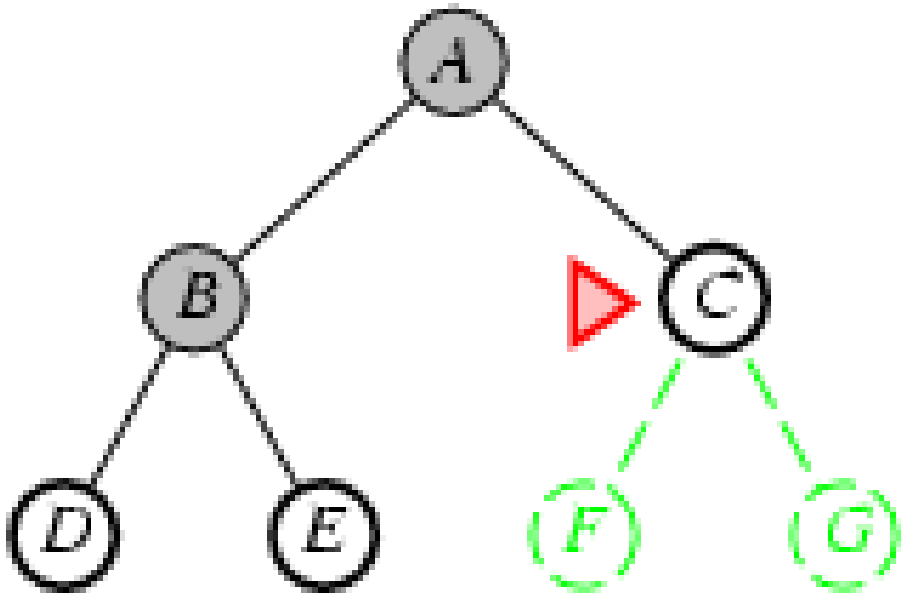


Breadth-first search

Expand shallowest unexpanded node

Implementation:

- *fringe* is a FIFO queue, i.e., new successors go at end

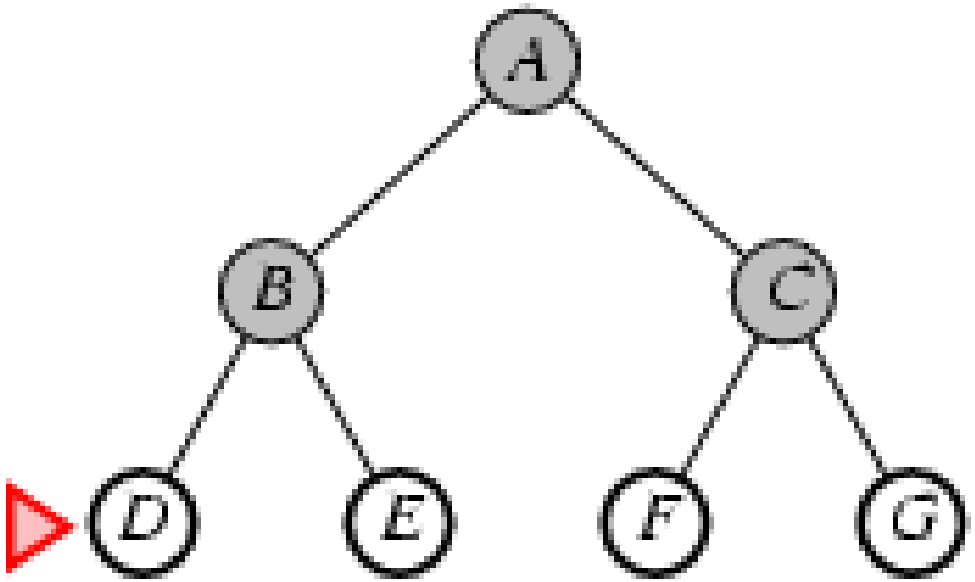


Breadth-first search

Expand shallowest unexpanded node

Implementation:

- *fringe* is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete

- Yes: if the shallowest goal node is at some finite depth *d*, BFS will find it after expanding all the shallower nodes, provided *b* is finite

Optimal

- Yes: BFS will always find the shallowest goal node. The shallowest node will be optimal if the path cost is a non-decreasing function of the depth of the node (for example, when all actions have the same cost)

Time

- In worst case bfs expands all but the last node at level d (goal level): $b+b^2+b^3+\ldots+b^d + b(b^{d+1}-b) = O(b^{d+1})$

Space

- Every node generated must remain in memory because it is either part of the fringe or is the ancestor of a fringe node. $\Rightarrow$ $O(b^{d+1})$

Breadth-fist search Example

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 mb
4	111,100	11 seconds	106 mb
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Example: b = 10, d varies, 10,000 nodes generated per second, and a node requires 1,000 bytes of storage.

Space is a biggest problem (more than time)

Uniform-cost search

BFS is optimal *when all step costs are equal*, because it always expands the shallowest unexpanded node.

Uniform-cost search algorithm is a variation of breadth-first, where we always expand the node with the lowest cost

Such algorithm is optimal because of the way it is designed

If all step-costs are equal, this is identical to breadth-first search.

Implementation:

- *fringe* = queue ordered by path cost

Uniform-cost search

Complete

- Yes: provided the cost of every step ≥ ε (ε = a small positive constant), this ensures that the algorithm will not get stuck in an infinite loop, constantly expanding a node that has a zero cost action.

Optimal

- Yes: provided the cost of every step ≥ ε (ε = a small positive constant). As nodes are expanded in increasing order of path cost the fist goal node selected for expansion is the optimal solution.

Uniform-cost search

Because UCS is guided by path-cost rather than depth, its complexity cannot easily be characterized in terms of b and d.

Instead let C* be the cost of the optimal solution, and assume that every action costs at least ε.

The algorithms worst-case time and space complexity is $O(b^{1+[C^*/\varepsilon]})$ which can be much greater than b^d.

This is because UCS can, and often does, explore large trees of small steps before exploring paths involving large and perhaps useful steps.

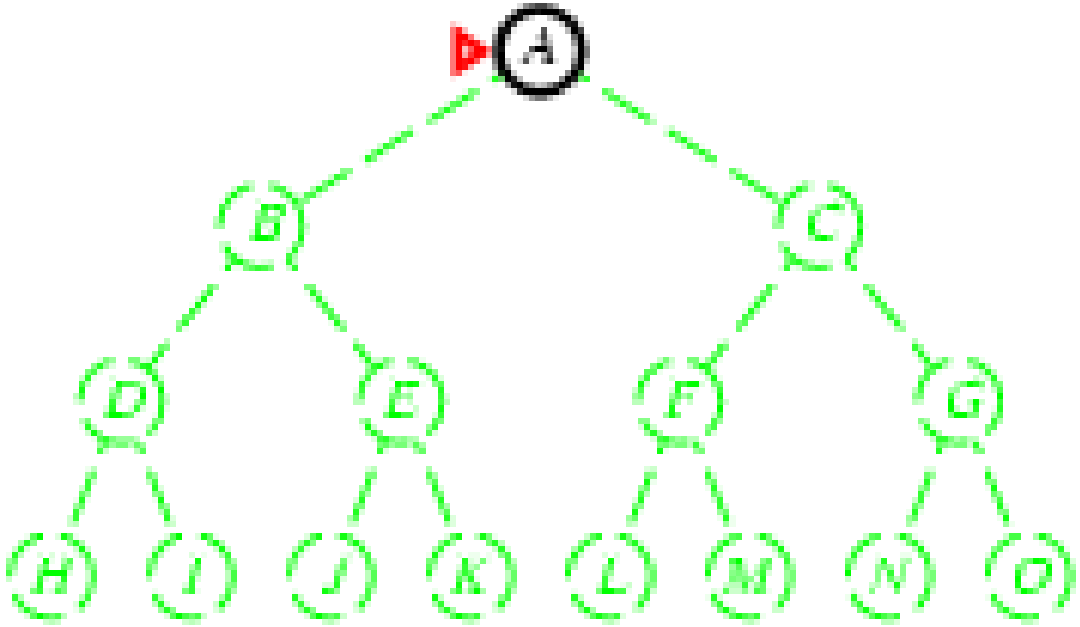
- Time number of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space number of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$

Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO (stack), i.e., put successors at front

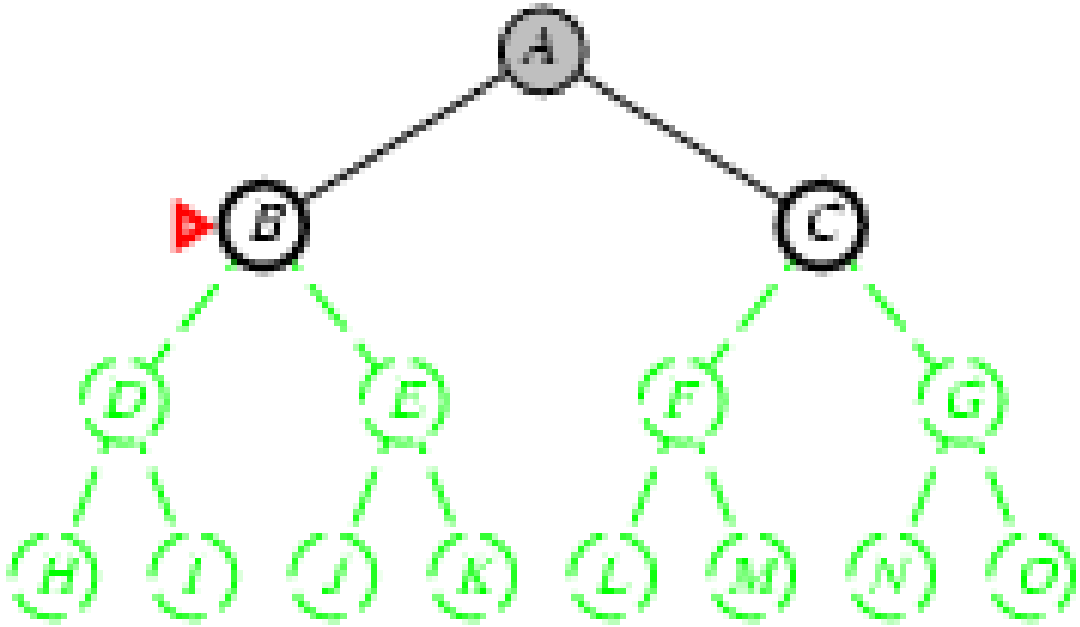


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

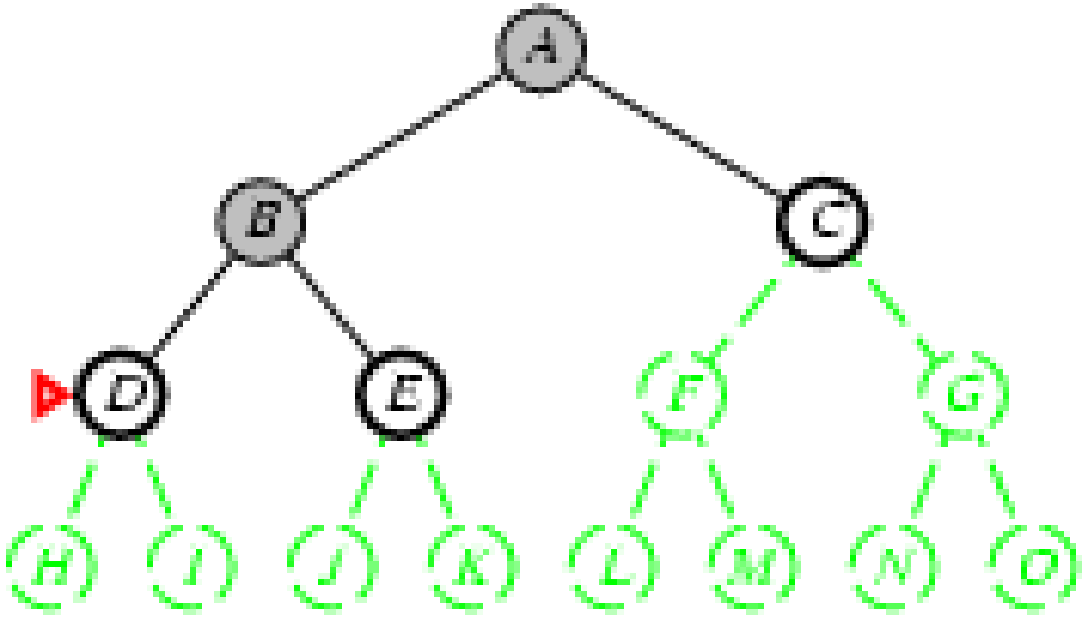


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

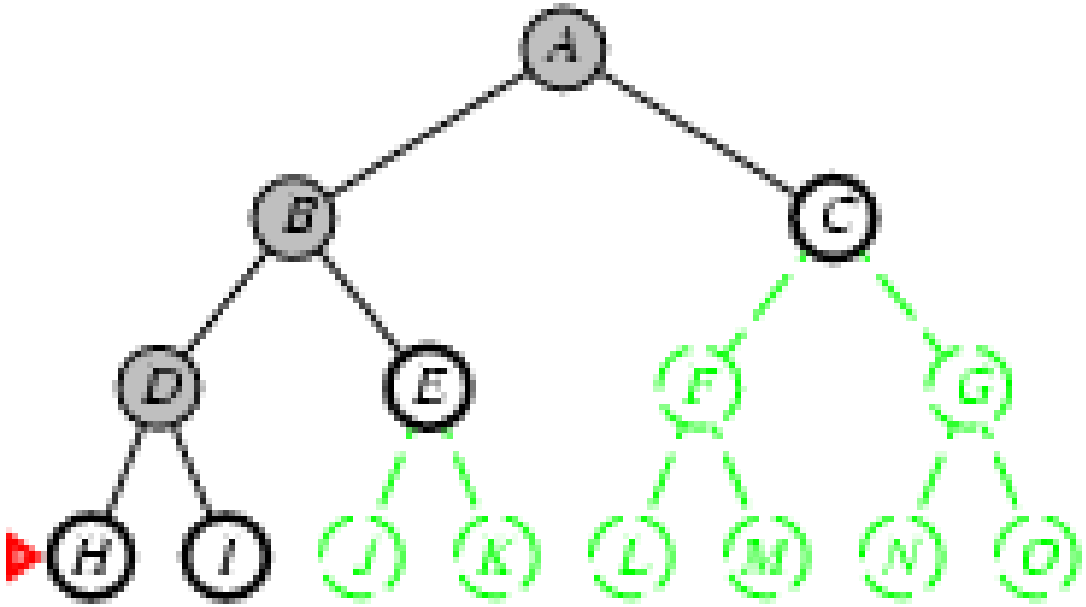


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

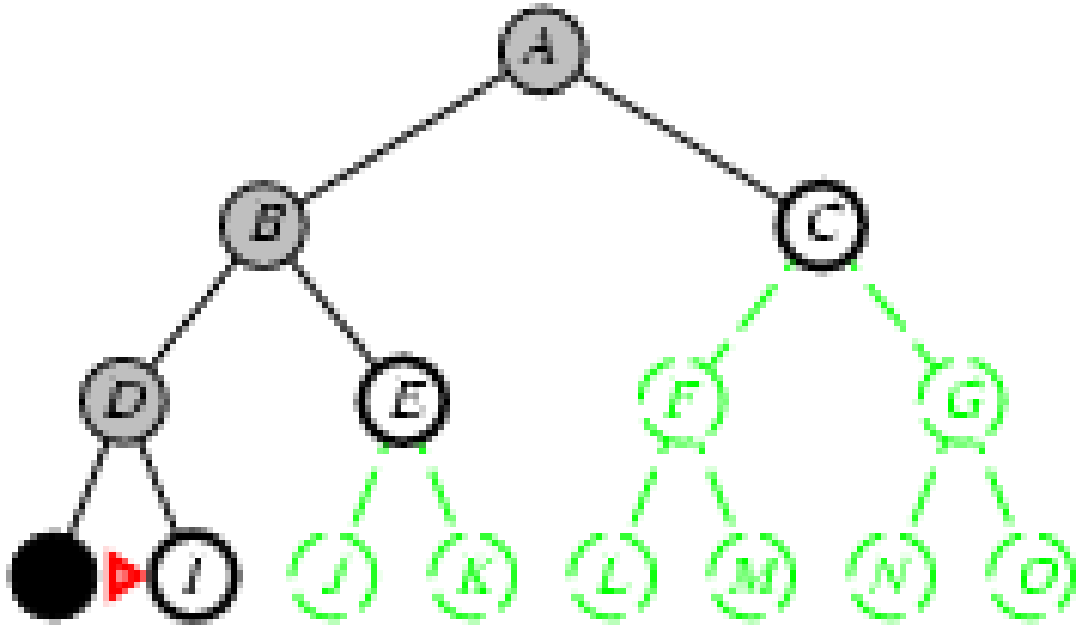


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

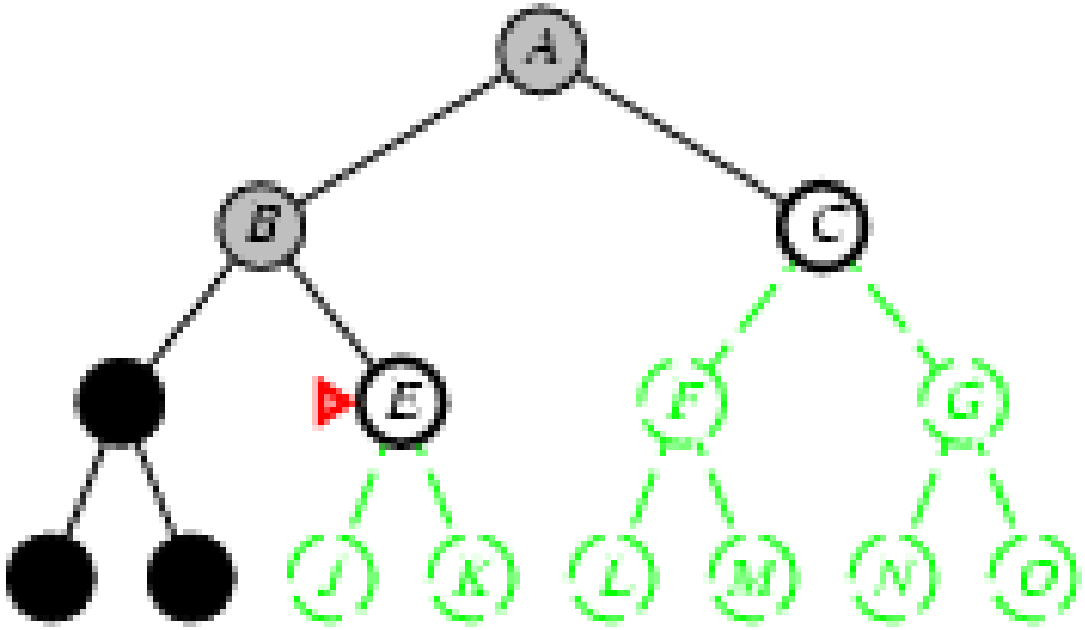


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

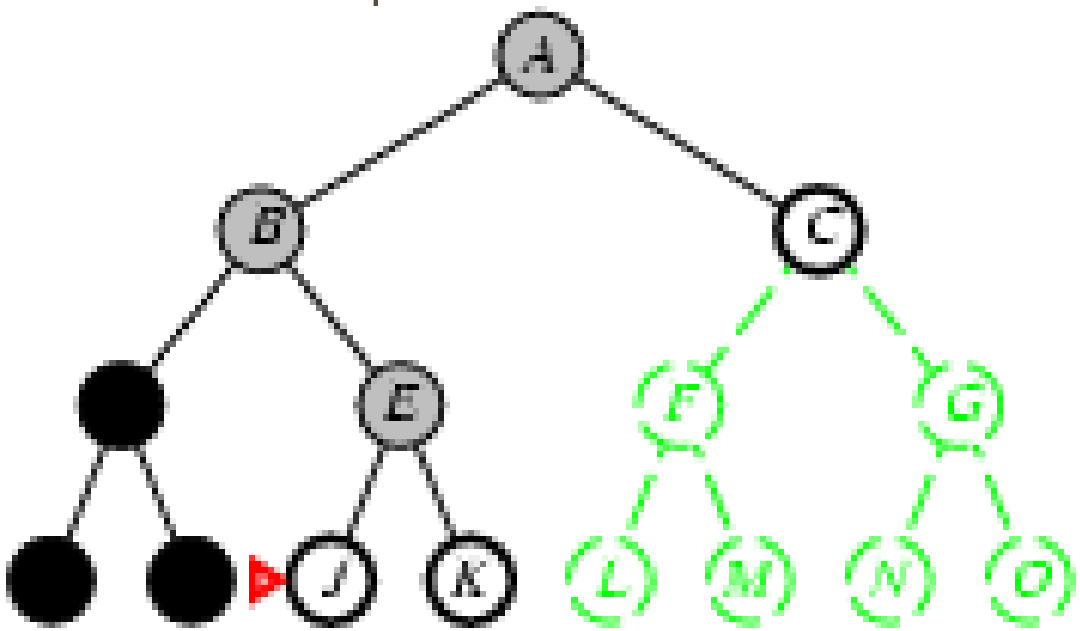


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

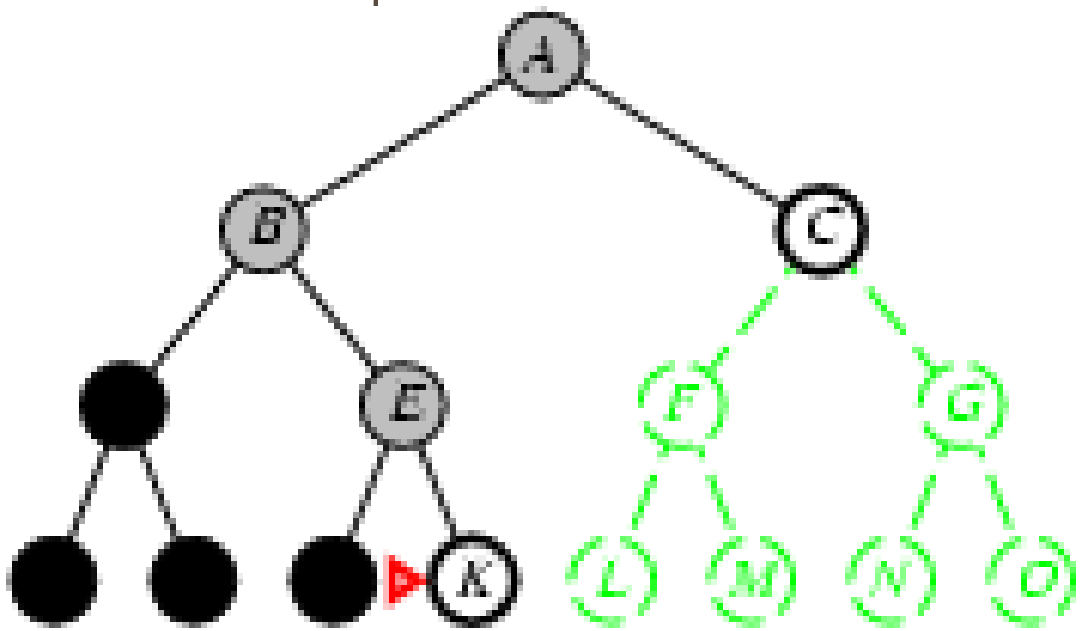


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

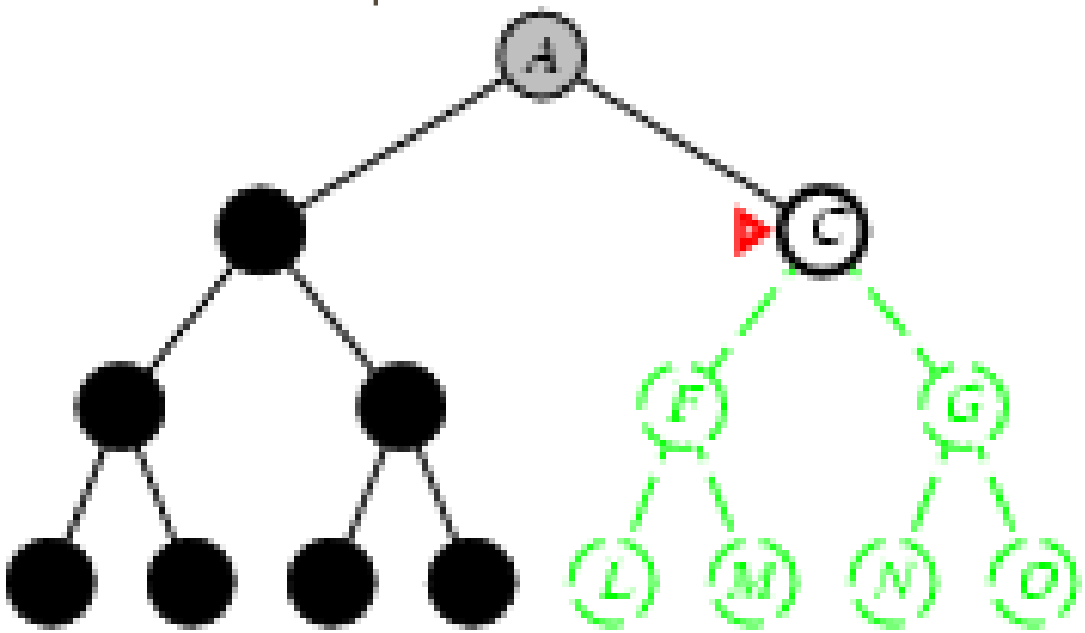


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

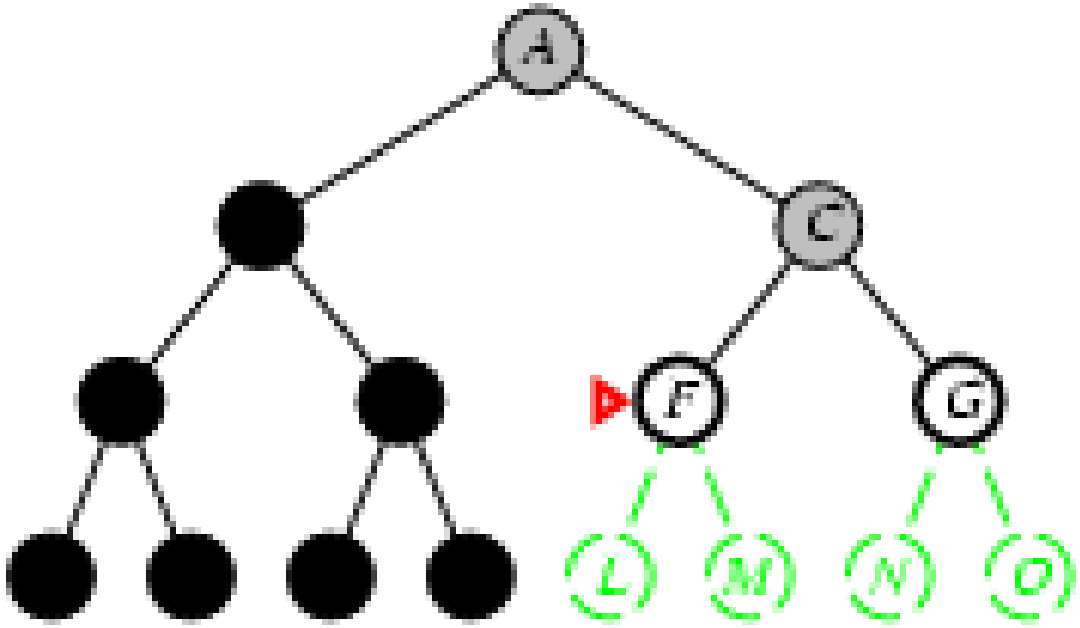


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

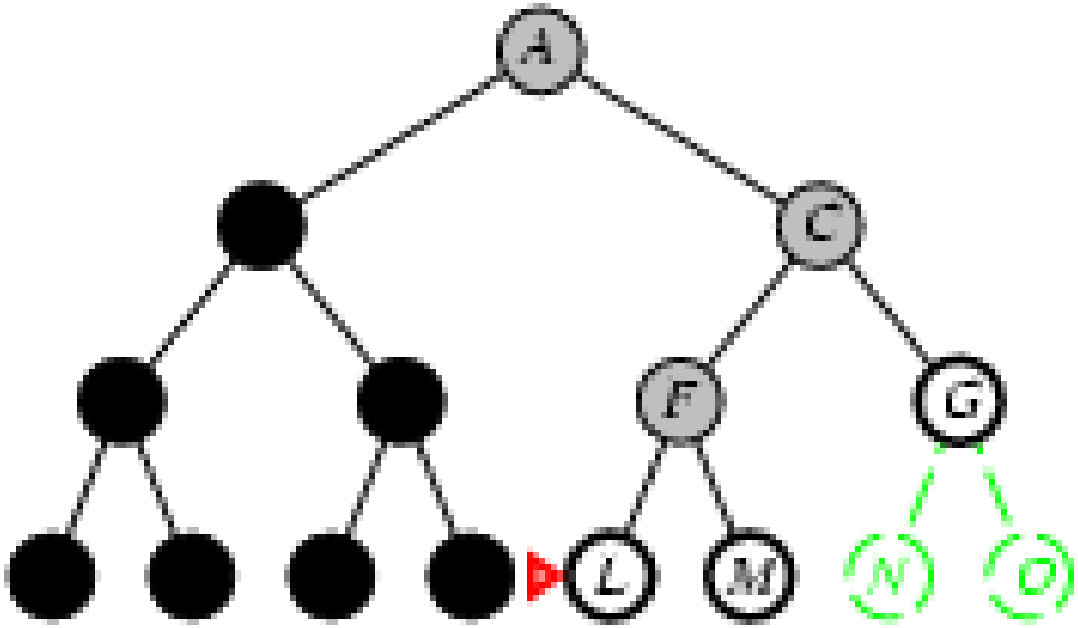


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front

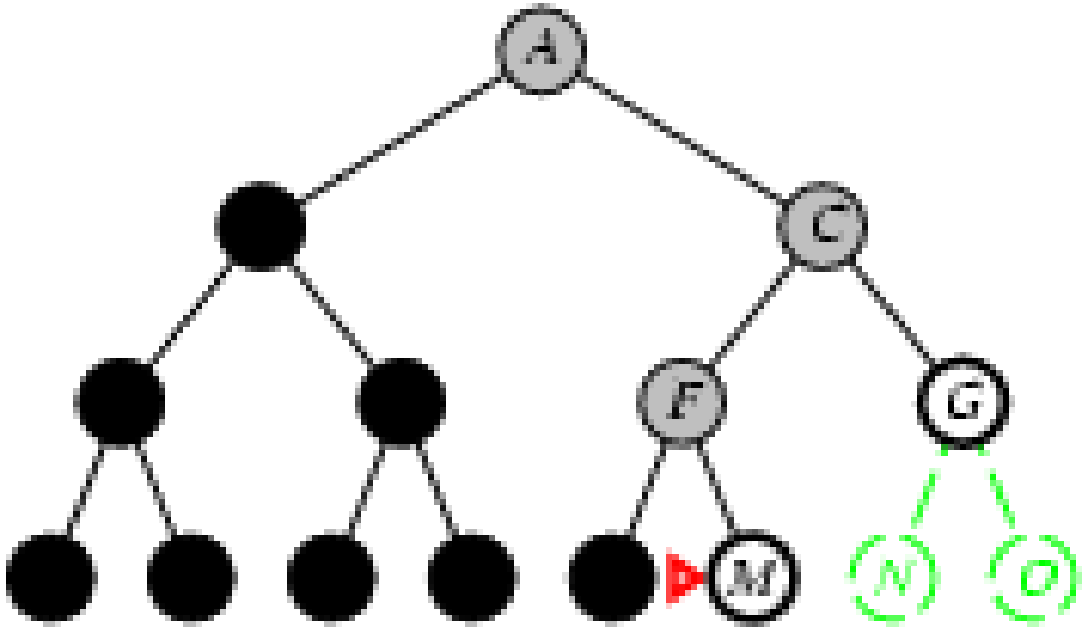


Depth-first search

Expand deepest unexpanded node

Implementation:

- *fringe* = LIFO, i.e., put successors at front



Properties of Depth-first search

Complete

- No: DFS can get stuck going down a very long (even infinite) path when a different choice would lead to a solution near the root of the search tree.
- Using the example presented in the last few slides, DFS would search all of the left of the search tree even if node C were the goal node. If the left subtree were of unbounded depth but contained no solutions, DFS would never terminate.

Optimal

- No: continuing with our analysis of the previous example, if node C and J were goal nodes, DFS would return the path to J as the solution!

Properties of Depth-first search

Time

- In worst case DFS generates all of the $O(b^m)$ nodes in the search tree, where *b* is the branching factors and *m* is the maximum depth of any node. Note that m can be much larger than d (the depth of the shallowest solution) and is infinite if the tree in unbounded.
- Summary: terrible if *m* is much larger than *d*, but if solutions are dense, may be much faster than breadth-first

Space

- Has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded it can be removed from memory as soon as all its descendants have been fully explored. (the black nodes on the previous slides). For a state space with branching factor b and maximum depth m, dfs requires storage of only *bm+1*

Depth-limited search

The DFS difficulties with unbounded trees can be alleviated by supplying DFS with a predetermined depth limit *l*. Nodes at depth *l* are treated as having no successors.

This depth limit avoids the dfs problems with unbounded trees but introduces another source of incompleteness: if we choose l < d. (This is not unlikely when d is not known).

DLS is also non-optimal if we choose l > d

Time complexity: $O(b^l)$ Space complexity: $O(bl)$

DFS can be viewed as a special case of DLS with $l = \infty$

Iterative deepening search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy that finds the best depth limit.

It does this by gradually increasing the limit -- first 0, then 1, then 2, … - until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.

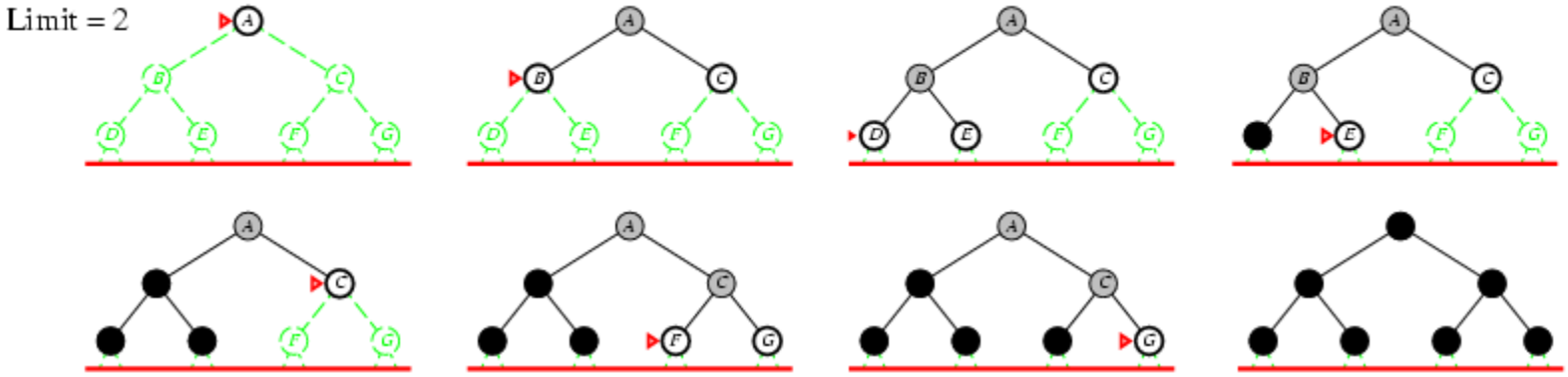
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening search l =0

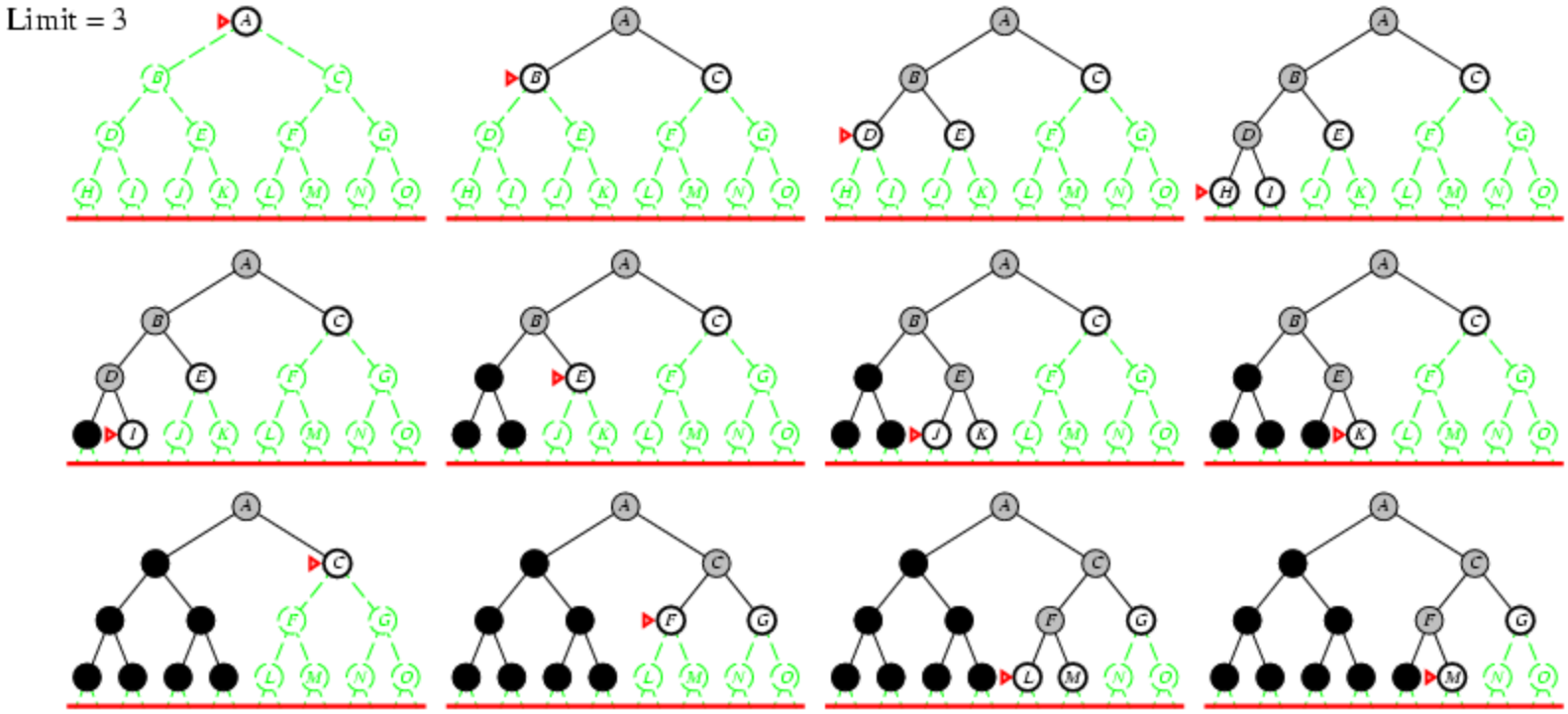
Limit = 0

Iterative deepening search l =1

Iterative deepening search l =2



Iterative deepening search l =3



Iterative deepening search

IDS may seem wasteful, because states are generated multiple times. However, it turns out that this is not very costly.

The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are generated at the bottom level, so it does not matter much that the upper levels are generated multiple times.

In IDS the nodes on the bottom level (depth d) are generated once, those on the next to bottom level are generated twice, etc. up to the children of the root which are generated d times.

So the total number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d)b + (d-1)\ b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

This gives us a Time Complexity of $O(b^d)$

Iterative deepening search

If we compare this with the nodes generated by a breadth-first search to depth *d* with branching factor *b*:

$$N_{BFS} = b + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d + b^{(d+1)} - b$$

Notice that BFS generates some nodes at depth d+1, whereas IDS does not. The result is that IDS is actually faster than BFS despite the repeated generation of states.

For $b = 10$, $d = 5$,

- $N_{BFS} = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + 999{,}990 = 1{,}111{,}100$
- $N_{IDS} = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$

Properties of Iterative deepening search

Complete

- Yes

Optimal

- Yes: if step cost = 1

Time

- $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space

- $O(bd)$

In general, iterative deepening search is the preferred uninformed search method when there is a large search space and the depth of the solution in not known.

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Repeated States

The possibility of wasting time by expanding states that have already been encountered and expanded before is one of the most important complications to be aware of in search.

For some problems the possibility never occurs, the state space is a tree and there is only one path to each state.

For other problems, repeated states are unavoidable. This includes all problems where the actions are reversible: route-finding, slide-blocks puzzles, etc

Repeated States

Repeated states, can cause a solvable problem to become unsolvable if the algorithm does not detect them.

Detection usually means comparing the node about to be expanded to those that have been expanded already; if a match is found, then the algorithm has discovered two paths to the same state and can discard one of them.

Repeated States

"Algorithms that forget their history are doomed to repeat it."

Graph-search: we can modify the general tree search algorithm to include a data structure called closed list, which stores every expanded node.

- The fringe of unexpanded nodes is sometimes called the open list.

If the current node matches a node on the closed list it is discarded instead of being expanded.

Summary

Before an agent can solve a problem it must formulate a goal and then use the goal to formulate a problem.

A problem consists of four parts: the initial state, a set of actions (successor function), a goal test function and a path cost function.

The environment of the problem is represented by a state space. A path through the state space from the initial state to a goal state is a solution.

Node ≠ state

State space ≠ search tree/graph

Variety of uninformed search strategies

Search algorithms are judged on the basis of completeness, optimality, time complexity and space complexity.

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Questions?